# A Runnable Functional Formal Memetic Algorithm Framework

Natalio Krasnogor  Pablo A. Mocciola*  David Pelta  German E. Ruiz
Wanda M. Russo†

LIFIA, Departamento de Informática, Universidad Nacional de La Plata.
C.C.11, Correo Central, 1900, La Plata, Buenos Aires, República Argentina.
E-mail: {natk,pablom,davp,gruiz,wanda}@info.unlp.edu.ar
URL: http://www-lifia.info.unlp.edu.ar/

**Abstract**

Historically, 'Functional Programming' (FP for short) [1] has been associated with a small scope of applications, mainly academic. The computer science community did not pay enough attention to its potential, perhaps due to the lack of efficiency of functional languages. Now, new theoretical developments in the field of FP [11] are emerging, and better languages (e.g. Haskell [21], Concurrent and Parallel Haskell [22, 14]) have been defined and implemented.

Genetic algorithms (GA) are search and optimization techniques which work on a nature inspired principle: the Darwinian evolution. The corner idea of Darwin theory is that of natural selection. The concept of natural selection is captured by GA. Specifically, solutions to a given problem are codified in the so called chromosomes. The evolution of chromosomes due to the action of crossover, mutation and natural selection is simulated through computer code. GA have been broadly applied and recognized as a robust search and optimization technique. GA combined with a local search stage were called "Memetic Algorithms" after [17].

In this paper a functional framework for formal memetic algorithms [23] is introduced. It can be easily extended, by subclassification of the class hierarchy, to provide genetic algorithm specialization (memetic algorithm, genetic algorithm with islands of possible solutions, etc) and additional genetic operators' behavior. To run the framework over a particular problem, a proper encoding of chromosomes should be provided with an instantiation of the genetic operators. We claim that functional programming languages, at least the one in which our framework has been developed (Haskell), have reached the necessary maturity to deal with combinatorial optimization problems.

**Keywords:** Functional Programming, Memetic Algorithm, Combinatorial Optimization.

*Pablo Mocciola is partially supported by "Fundación Antorchas"
†Authors appear alphabetically

# A Runnable Functional Formal Memetic Algorithm Framework

## 1 Introduction

Historically, 'Functional Programming' (FP for short) [1] has been associated with a small scope of applications, mainly academic. The computer science community did not pay enough attention to its potential, perhaps due to the lack of efficiency of functional languages. Now, new theoretical developments in the field of FP [11] are emerging, and better languages (e.g. Haskell [20], Concurrent Haskell [22]) have been defined and implemented. Also, the gap between theory and practice is smaller in this paradigm than that of other paradigms, making FP a good choice for developing simulation and optimization programs [25]. Traditionally, all programs for optimization problems have been written in C, C++ or Ada; this builds a firewall between developers and end-users.

Genetic Algorithms are suitable to be modeled with a lazy concurrent functional language for many reasons:

- non-computer-science people can think in a very high abstraction level and map their ideas, almost directly, to functional pseudo-code;
- the learning curve of a FP language is smoother than that of an imperative one, bridging the gap between developers and users;
- functional code is concise;
- many optimization processes are intrinsically parallel and FP is specially adequate for managing parallelism;
- the use of lazy languages avoids the construction of many feasible solutions until they are needed (if ever);

The paper is organized as follows. Section 2 recalls a short review of genetic and memetic search. Section 3 is devoted to some related approaches which have greatly influenced our work. In section 4 we show the framework's system definition and architecture and show how to implement it and instantiate a problem encoding. Two NP optimization problems are showed as tests bed for our framework in section 5. Finally, section 6 is devoted to conclusions and future works.

## 2 Genetic and Memetic Search, a Short Review

Genetic algorithms (GA) are search and optimization techniques which work on a nature inspired principle: Darwinian evolution. In 1859, the first edition of "On the origin of species by means of natural selection, or the preservation of favoured races in the struggle of life" appeared. Shortly known as "The origin of species", Darwin's work became one of the most read book of the century. The corner idea of Darwin's theory is that of natural selection. As species give rise to many more individuals than could survive, any being that shows a variation that is favorable to its survival will be "naturally selected" and because of inheritance every selected being will spread its new and modified shape [3]. The concept of natural selection is captured into GA. Specifically, solutions to a given problem are codified in the so called chromosomes. The evolution of chromosomes due to the action of crossover, mutation and natural selection are simulated through computer code. GA have been broadly applied and recognized as a robust search and optimization

technique. For a survey of GA, Memetic Algorithms(MA) and a comprehensive list of referred papers browse at [16].

In [9], J. Holland, introduced to the scientific community the genetic algorithms as are known today. Basically, it consists of a population that is a set of chromosomes. Each chromosome is formed by a sequence of genes. Originally genes, and hence chromosomes, were sequences of bits, where each sequence of bits represented a solution to a "part" of a given problem or the specification of some feature of the individual codified by the chromosome. Let us clarify this with an example. Suppose we are dealing with the following NP optimization problem[1]:

**Definition 1** *Maximum Constrained Partition(MCP)*

- **INSTANCE:** *Finite set $A$ and a size $s(a) \in \mathcal{Z}^+$ for each $a \in A$, element $a_0 \in A$, and a subset $S \subseteq A$.*
- **SOLUTION:** *A partition of $A$, i.e., a subset $A\prime \subseteq A$ such that $\sum_{a \in A\prime} S(a) = \sum_{a \in A - A\prime} S(a)$.*
- **MEASURE :** *Number of elements from $S$ on the same side of the partition as $a_0$.*

and we wish to use genetic algorithms to find near optimal solutions. First, an encoding must be found, that is, we need to specify how a solution will be represented within a chromosome. For example we may want to associate a bit array to a partition, where position $i$ in the array is zero if the $i^{th}$ element of $A$ is in the first partition or a non-zero value otherwise.

If we have the following instance of *MCP*:

- $A = \{12, 2, 5, 10, 3, 2, 4, 6\}$
- $s(x) = id(x) \forall x \in A$
- $a_0 = 5$
- $S = \{2, 3, 6, 12\}$

In this case a chromosome can be represented as a sequence of eight bits, i.e., $Cr_1 = (0, 1, 1, 1, 1, 1, 0, 0)$ means that the two partitions are $A = \{12, 4, 6\}$ and $A\prime = \{2, 3, 5, 10, 2\}$ where $\sum_{a \in A} s(a) = \sum_{a \in A} id(a) = \sum_{a \in A} a = 22$ and $\sum_{a \in A\prime} s(a) = \sum_{a \in A\prime} id(a) = \sum_{a \in A\prime} a = 22$. The quality of the solution $Cr_1$ or its "fitness" value, as one can suspect, is given by the measure as stated in the problem definition, $f(Cr_1) = 2$. We should mention that part of the success applying GA to real life applications is due to the fact that no one needs to know how to construct a solution, furthermore, a way to describe how a solution looks like is needed. GA are blind symbolic optimizers that operate over "building blocks". If we are interested in finding near optimal configurations we only need to provide a way in which solutions, if found, could be easily evaluated and recombined, that is, we are assuming that problems are decomposable. Now, imagine that we do not only have this feasible solution, $Cr_1$, but also a set of chromosomes, a population of size $n$, $P = \{Cr_1, \ldots, Cr_n\}$. To generate better solutions from this original set we will need to provide mechanisms to "search" the conformational space of the problem. In essence there are three alternatives. The first one is to randomly change some or all genes of a chromosome, and this is what mutation is about. The second choice is recombination. Given $t$ chromosomes recombine them in some way producing $r$ new chromosomes. This process is called crossing over. The last one is local search. We can allow each chromosome

[1] We are following [2] format for problem presentation

to enhance itself with a stage of local search[2]. Standard GAs make use of mutation and crossover, while more sophisticated applications use local search stages also. GA combined with local search were named "Memetic Algorithms" after [17]. In our example, $P = \{Cr_1 = (0,1,1,1,1,1,0,0), \ldots, Cr_k = (0,1,1,0,1,1,1,1), \ldots\}$. In this way, chromosome $Cr_1$ induces partitions $A_1 = \{12,4,6\}, A_1\prime = \{2,3,5,10,2\}$ and $Cr_k$ partitions $A_k = \{12,10\}, A_k\prime = \{4,5,2,3,2,6\}$ where $f(Cr_1) = 2 < f(Cr_2) = 3$. From $Cr_1$ and $Cr_k$ it is possible to produce $Cr_{(1,k)} = (0,0,1,1,1,0,1,0)$ as an offspring taking genes $1,3,4,5$ and 7 from the first parent and the others from the second one. When $Cr_{(1,k)}$ is evaluated, $f(Cr_{(1,k)}) = 1$, its fitness happens to be worse than its ancestors'. Because probabilities of survival and mating are proportional to fitness, it is unlikely that this new chromosome will spread into the population. The mutation process can be viewed as complementing a given number of bits in a chromosome, that is, mutation puts zero where there was a one and a one where a zero was found. Artificial selection operates over the entire population once mutation and crossover, eventually local search, were already applied. There are a number of different ways to implement it, but it can be viewed as a process that operates over a set composed by the original population, whose size was $n$, and the set of offsprings of size $k$. From this set of $n+k$ chromosomes, the best $n$ are selected, rejecting the other ones.

Now that we have already introduced the main concepts of GA, we are in position to give a sketch of the "canonical genetic algorithm".

```
[1] Start with a randomly generated population
    of chromosomes ( e.g., candidate solutions
    to a given problem ).
[2] Repeat until finalization_criteria is met
 [2.1] Apply genetic operators (crossover and
       mutation) to the population.
 [2.2] Calculate the fitness of each chromosome in
       the population.
 [2.3] ** if MA, run local search with each
          chromosome **
 [2.4] Apply selection and get a new population.
[3] Output best chromosome.
```

The above pseudocode also belongs to a Memetic Algorithms when the line with the '**' is instantiated.

GA and MA were applied in a number of different areas, i.e., optimization, automatic programming, machine and robot learning, economic models, immune system models, ecological models, populations genetic models, interaction between evolution and learning and models of social systems. For a survey refer to [15].

# 3 Some Related Work

In this section we comment on three different approaches, with their own "pros & cons", which have greatly influenced our work. Nevertheless, space precludes describing every proposal in detail. These works were chosen because, in some sense, they are representatives of the sort of work that have been done within the GA community to provide GA tools.

## 3.1 Genesis

In [8] the **Gene**tic **S**earch **I**mplementation **S**ystem (GENESIS Version 5.0) was described. This system was written to promote the study of genetic algorithms for solving generic

[2] In this case we departure from Darwinian evolution and we enter Lamarck's Kingdom

problems that involve function minimization. Grefenstette's work considers genetic algorithms as task independent optimizers. Under this design, users should only provide an "evaluation" function (fitness). Chromosomes are represented by bit strings where an alternative individuals' representation using vectors of real numbers (user-level representation) might be provided. We have found that using GENESIS with nontrivial problems becomes very difficult mainly due to the constraint in the representation of solutions. Also,the system is quite difficult to extend and modify.

### 3.2 Genetic algorithms in Haskell with polytypic programming

Vestin's master thesis [24] presents a genetic algorithm in Haskell [20] that uses polytypic programming [10] and Haskell's class system [13]. Due to the use of polytypism, the program is easily extended to solve new problems using the same algorithm. A powerful representation-free GA is implemented. In this work crossover and mutation are programmed via polytypic functions. Polytypic functions are defined by induction on the structure of data types. There is only one function definition, but instances for every data type on which it is applied. In this way, Vestin's GA can be applied to a plethora of chromosomes representations without changing anything in the original GA. We will discuss later some limitations to Vestin's work.

### 3.3 Formal Memetic Algorithms

The purpose of the work presented in [23] is to formalize memetic algorithms and to provide a unified algebraic framework for considering both memetic and genetic algorithms. The authors define a set of representation-independent operators and, as test bed, they optimize over TSP instances. Radcliffe's work is a bridge between imperative implementation of GA and functional ones. The referenced work defines a representation function $\rho : \mathcal{S} \mapsto \mathcal{C}$, where $\mathcal{S}$ is the space of solutions to a given problem and $\mathcal{C}$ the set of chromosomes. The fitness is defined as a function $f : \mathcal{C} \mapsto \Re^+$. A *hill-climber* operator $\mathcal{H}$ is defined in terms of the representation space $\mathcal{C}$, a move operator $\mathcal{Q}$ and the subspace of $\mathcal{C}_\mathcal{Q}$ of local optimal under the move operator $\mathcal{Q}$, that is, $\mathcal{H} : \mathcal{C} \times \mathcal{K}_\mathcal{H} \mapsto \mathcal{C}_\mathcal{Q}$, where $\mathcal{K}_\mathcal{H}$ is a control set. Crossover is characterized as a function that takes two chromosomes, a control set and returns a chromosome, $\mathcal{X} : \mathcal{C} \times \mathcal{C} \times \mathcal{K}_\mathcal{X} \mapsto \mathcal{C}$. The mutation is a function of the type $\mathcal{M} : \mathcal{C} \times \mathcal{K}_\mathcal{M} \mapsto \mathcal{C}$. A genetic algorithm is then defined as a reproductive function of type $\mathcal{R}_g : \mathcal{C} \times \mathcal{C} \times \mathcal{K}_\mathcal{M} \times \mathcal{K}_\mathcal{X} \mapsto \mathcal{C}$ where $\mathcal{R}_g(x, y, k_\mathcal{M}, k_\mathcal{X}) \stackrel{\text{def}}{=} \mathcal{M}(\mathcal{X}(x, y, k_\mathcal{X}), k_\mathcal{M})$. If a hill-climber is defined and used the memetic reproductive plan becomes $\mathcal{R}_m : \mathcal{C}_\mathcal{G} \times \mathcal{C}_\mathcal{G} \times \mathcal{K}_\mathcal{H} \times \mathcal{K}_\mathcal{M} \times \mathcal{K}_\mathcal{X} \mapsto \mathcal{C}_\mathcal{G}$ being $\mathcal{R}_m(x, y, k_\mathcal{H}, k_\mathcal{M}, k_\mathcal{X}) \stackrel{\text{def}}{=} \mathcal{H}(\mathcal{M}(\mathcal{X}(x, y, k_\mathcal{X}), k_\mathcal{M}), k_\mathcal{H})$. With this definitions we can try to develop a "runnable" formal memetic algorithm, and this goal will be reached using a functional language.

## 4 System Definition and Architecture

Our research is focused in developing a runnable functional GA framework for finding approximate solutions to combinatorial problems; the framework should provide:

- A tool for developing combinatorial optimization applications in which the optimization core can be built as an instance of a genetic algorithm.
- The instantiation of various kind of crossovers, mutations, selection procedures, local search heuristics, etc. These features provide a solid ground for experimentation on the GA field *per se*.

- The ability to test a given problem under different kind of individuals' encoding and alternative genetic operator.
- The possibility to choose between an isolated population structure and population islands with migration operators, being synchronous or asynchronous.

The architecture of the application consists of three main modules: genetic algorithm class hierarchy, population module, and problem module. The first one determines a hierarchy of specific genetic algorithms. Each particular class of genetic algorithms is specified with two sets of operations. One set is user-dependent and defined according to the particular problem to be solved. The other set contains default functions which can be redefined to specify alternative behavior. The population module is an abstract data type that encapsulates representation and functions about the population management. A population is a collection of genomes or individuals. When the GA is to be used a problem must be specified in terms of its own parameters, GA parameters and an appropiate encoding. In figure 1 an object-like chart of the system is presented.

## 4.1 Implementation Issues

The framework is under development in Haskell [20], a pure functional programming language that supports high order functions, lazy evaluation, class system and modules. It is worth to mention that the first prototype of our system was built using the Hugs interpreter [12]. In what follow we will briefly describe the main aspects of our implementation.

### 4.1.1 Population Module

Because genetic algorithms are continously working with the population it should be carefully implemented. It is codified by an abstract data type with the following representation and operations respectively[3]:

```
data Population a = Pop (Array Integer a)

newPopulation :: [a] -> Population a
addPopulation :: Population a -> [a] -> Population a
individual    :: Population a -> Integer -> a
totalFitness  :: Population a -> b
best          :: Population a -> b
average       :: Population a -> Double
size          :: Population a -> Integer
forAll        :: (a -> IO a) -> Population a -> IO (Population a)
```

### 4.1.2 Genetic Algorithms Class Hierarchy

A genetic Object "knows" how to:

- Compute fitness.
- Set up the matting pool.
- Generate offsprings.
- Produce mutations.

[3] contexts were omited

which are user-dependent operations.

The hierarchy of possible genetic algorithms is structured under Haskell's class system where the main class is `GenObject`; it specifies the basic genetic algorithm's behaviour and problem templates. Nevertheles the above functions are ment to be provided by the user, the main operations of the genetic algorithm skeleton are given by the default implementation. A sketch of the code is presented below.

```
class GenObject a where
 --given by the user
 fitness        :: Num b => a -> b
 newGenObj      :: IO a
 matingPool     :: Population a -> IO [[a]]
 cross          :: [a] -> IO a
 mutate         :: a -> IO a
 --default
 initialization :: Int -> IO (Population a)
 crossover      :: Double -> Population a -> IO (Population a)
 mutation       :: Double -> Population a -> IO (Population a)
 selection      :: Int -> Population a -> IO (Population a)
 geneticAlg     :: Params -> IO(Population a)
```

Each kind of genetic algorithm specialization (memetic algorithm, genetic algorithm with islands of possible solutions, etc) is implemented by specialization of the hierarchy class.

The genetic algorithm body follows basically the steps presented above as "canonical genetic algorithm":

```
geneticAlg     :: Params -> IO (Population a)
geneticAlg params =
  do let i = iterations params
         sizePop = sizePopulation params
     pop <- initialization sizePop
     doit i pop
       where doit n pop =
               if n==0 then return pop
                 else
                   let xPro    = xProbability params
                       mPro    = mProbability params
                       sizePop = sizePopulation params
                   in
                       crossover xPro pop    >>=
                       mutation mPro         >>=
                       selection sizePop     >>=
                       doit (n-1)
```

### 4.1.3 Problem Module

To run the framework over a particular problem, a proper encoding of chromosomes should be provided with an instantiation of genetic operators. As an example, consider the specialization shown in figure 1 where $opPR_i$ is a particular implementation of the genetic operator $op_i$. In that way, two different GA instances were derived, one for each problem and operation definitions. The next few lines of code are just a template example.

```
data ProblemRepresentation = ....

instance GenObject ProblemRepresentation where
```

```
fitness         = fitnessPR
newGenObj       = newGenObjPR
cross           = crossPR
matingPool      = matingPoolPR
mutate          = mutatePR
```

#### 4.1.4 Random Module

A critical subsystem is the random module. For the first version of the framework, implemented in Hugs, we have used Hugs' random library which is the same as the one in the standard Haskell's libraries. The functions were not as efficient as we needed, because genetic algorithms make extensive use of random functions. As a consequence we are doing research in efficient functional random libraries, and in order to implement them, we are using foreign-language interfaces such as Green Card [18].

## 5 Two NP Optimization Problems as Tests Bed

In this section we will define two NP optimization problems that will be used as test beds for our framework. Both of them have important applications both in theory and practice. Also, we will report some data on the use of the functional genetic framework in these problems. We should mention that in this paper our goal is to test the reliability of the framework and its usability (features) on different problems. We are not looking for "better-than" nor "faster-than" results by means of a comparison with other approaches to these NP optimization problems.

### 5.1 Problem Definitions

**Definition 2** *Minimum Number Partitioning(MNP)*

- **INSTANCE:** *Sequence* $A = (a_1, a_2, \ldots, a_n), a_i \in \mathcal{Q}^+$.
- **SOLUTION:** *A sequence* $S = (s_1, s_2, \ldots, s_n)$ *of signs* $s_i \in \{-1, +1\}$.
- **MEASURE :** $u \equiv \mid \sum_{i=1}^{i=n} s_i * a_i \mid$.

Intuitively, the problem consists in finding a partition of the sequence $A$ into two subsets such that the sum of elements in each one is more or less the same. The problem *Partition* is a special case when $u = 0$. It was shown NP- complete by R.Karp in 1972. The importance of MNP as a test bed is based in the fact that many decision problems are reducible to MNP decision version, for example, *Bin Packing*, *Knapsack*, *Open-Shop Scheduling* and others. This problem has been approached using simulated annealing, genetic algorithms, neural nets and approximation algorithms. For an interesting comment on MNP see [5].

**Definition 3** *Maximum Betweenness(MB)*

- **INSTANCE:** *Finite set A, collection C of ordered triples* $(a, b, c)$ *of distinct elements from A*
- **SOLUTION:** *a one-to-one function* $f : A \mapsto [1 \ldots \mid A \mid]$.
- **MEASURE :** *Number of triples where either* $f(a) < f(b) < f(c)$ *or* $f(c) < f(b) < f(a)$.

One of the main goals of the *Genome Project* is to sequence and map the whole human genome. To accomplish that work, DNA is cut into pieces named clones. Each clone is studied and replicated. The aim of the mapping stage is to reconstruct the original placement of each clone within the complete DNA strand. In [7] it is shown that two simple models of physical mapping are NP- Complete reducing one of them to MB, which in turns was shown to belong to this class by Opatrny [19].

## 5.2 Results

We have instantiated the genetic framework to solve the problems described above. We have run two sets of experiments; the first one using Hugs interpreter for Windows 95 operating system in a PC pentium processor. The second set of experiments were done on a Sun Solaris operating system within a Sun Sparc Station 20. In the sequel we will comment about these two experimental settings. It is worth noting that the implementation done on the PC was a prototype.

### 5.2.1 Hugs Prototype

Figure 2 shows results for MNP over the following instance [1,2,3,4,5,6,7,8,9,10,1,1,1,1, 1,1,1], whereas figure 3 and figure 4 display data for MB. The instances for MB were: instance 1 = [('a','b','c'), ('f','a','d'), ('c','j','l'),('a','c','h'), ('c','a','j'), ('g','h','i')] and instance 2 = [('f','e','a'), ('b','g','m'), ('c','f','d'), ('e','a','d'), ('h','m','l'), ('b','c','h'), ('e','b', 'j'), ('g','h','i')].

Crossover and mutation probabilities were fixed in 0.75 and 0.1 respectively. Simulations were run using various population sizes until a fixed number of generations was reached.

For the intance of MNP problem the optimal solution is 0. The table shows that the optimal values could be reached with a small population size and few generations. It has to be noted that, in one case, the optimal solution appeared in the random initial population (generation 0).

In the second problem, the optimal values are not known for these instances but a naive upper bound is the number of *triples*, 6 in the first case and 8 in the second. As it is expected, best solutions (for equal populations sizes) could be reached with higher number of generations.

As a preliminary performance test we include the table presented a column with the number of reductions made by the Hugs interpreter. Note that the reductions' number scales linearly with population size. In order to show the ease of use, it must be considered that only a few changes in the first problem GA code were needed to deal with the second problem.

### 5.2.2 Haskell Implementation

In this section a time analysis is done by comparing our application with two implementations (in Haskell and C language) of exhaustive search algorithms.

The simulations were run over intances of the MNP problem with sequences of 15, 20, 25, 27 and 30 random elements with values in the range $1 \ldots 50$. In this case, the crossover and mutation probabilities of the GA were fixed in 0.60 and 0.05 respectively, whereas the population size and generation number were set up 30 and 50 respectively.

A time comparison among the three programs is showed in figure 5. It has to be noted that in the figure, the time's scale is measured in seconds and using a logarithm scale. As we expect, both exhaustive algorithms rise exponentialy with the instance size, whereas the time of GA remains stable below 6 seconds and depends only on the population size.

We conclude that the C version of the exhaustive algorithm is worthwile for intances with 20 or less elements, while our Haskell framework should be used for bigger instances.

## 6 Conclusions

We have used Haskell's class system to develop a framework of functions that implements the basic functionality of a GA. In contrast with other approaches, we allow the user to choose chromosomes encoding and the way crossover, mutation and selection operates (provided that the default behavior is not enough). In that way, a standard GA can be easily extended using the system class hierarchy.

The framework provides a base to experiment with evolutive algorithms that have arbitrary representation of population and operators. Moreover, it decreases the amount of work and coding needed for each new application by means of code reuse.

Goldberg in [6] states that GA are robust optimization methods because they reach the balance between efficiency and efficacy necessary for "survival" in many different contexts. Our framework introduces another dimension to GA tools, the ease of use, and a tight relation between formal memetic algorithms and practitioners.

## 7 Future Work

Many lines of research and future work remains to be done. Due to the fact that genetic algorithm operations could be thought of as concurrent processes that share resources (i.e. the genome population) we will work in a concurrent/parallel version of the framework . Moreover, we will use Vestin's idea to generalize genetic operators by polytyping. Vestin's work has the limitation that different kind of genetic operators are left aside, while our work is limited by not using the ideas of Vestin. We think that a powerful tool for GA experiencing could be made if we incorporate in our framework not only delegations (as it is already programmed) but also polytyping.

From this work we have learned that the functional formal memetic algorithm is a powerful tool to express and solve NP optimization problems in an easy way. Exhaustive profiling remains to be done having in mind: size of the intances, non-toy problems, memory used, speed up times, etc.

Also, the genetic algorithm framework is being tested in BioCom group[4] under different problems raised from combinatorial optimization and computational biology, in particular, the *Traveling Salesman Problem* and the *Protein Folding Problem*. Futhermore, an instance of the GA framework is being used in a genetic programming setting where rules of cellular automata are evolved to solve a certain task [4].

## References


[1] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[2] P. Crescenzi and V. Kann. A compendium of np optimization problems. URL `http://www.nada.kth.se/theory/problemlist.html`.

[3] C. Darwin. *El Origen de las Especies.* Ediciones Sarpe, 1983.

[4] E. De la Canal, N. Krasnogor, D. Marcos, D. Pelta, and W. Rissi. Encoding and crossover mistmatch in a molecular design problem. Accepted to present at The workshop of Artificial Intelligence in Design Conference, July 1998. Lisboa. Portugal.


[4] `http://www-lifia.info.unlp.edu.ar/~natk/~biocom`

[5] F. Díaz, A.and Glover, Hassan M. Ghaziri, M. Gonzáles, J.L.and Laguna, Moscato P., and Fan T. Tseng. *Optimización Heurística y Redes Neuronales - en Dirección de Operaciones e Ingeniería.*

[6] David. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, Inc., January 1989.

[7] M.C. Golumbic, H. Kaplan, and R. Shamir. On the complexity of dna physical mapping. *Advances in Applied Mathematics*, (15):251–261, 1994.

[8] J.J. Grefenstette. A user's guide to genesis: Version 5.0, October 1990.
URL `ftp://ftp.aic.nrl.navy.mil/pub/galist/src/genesis.tar.Z` .

[9] J.H. Holland. *Adaptation in Natural and Artificial Systems.* MIT press, Cambridge, MA, second edition, 1992.

[10] J. Jeuring and P. Jansson. *Polytypic programming*, pages 68–114. Number 1129 in LNCS. Springer-Verlag, 1996.

[11] Johan Jeuring and Erik Meijer, editors. *Advanced Functional Programming, LNCS 925.* Springer-Verlag, May 1995.

[12] Mark P. Jones. Hugs 1.3 - the Haskell user's Gofer system. User manual. Technical report, Department of Computer Science, University of Nottingham, August 1996.

[13] M.P. Jones. Functional programming with overloading and higher-order polymorphism. *Advanced Functional Programming,*, LNCS 925, May 1995.

[14] H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, March 1998.

[15] M. Mitchell and S. Forrest. Genetic algorithms and artificial life. Technical report, Santa Fe Institute. Working Paper 93-11-072, to appear in Artificial Life.

[16] P. Moscato. Memetic algorithms' home page.
URL `http://www.densis.fee.unicamp.br/~moscato/memetic_home.html`.

[17] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts:towards memetic algorithms. Technical report, CalTech Concurrent Computation Program Report 826, CalTech, Pasadena CA, 1989.

[18] T. Nordin and SL Peyton Jones. Green card: a foreign-language interface for haskell. In *In the Proceedings of the Haskell Workshop*, Amsterdam, June 1997.

[19] J. Opatrny. Total ordering problems. *SIAM J. Computing*, 8(1):111–114, 1979.

[20] John Peterson, Kevin Hammond, et al. Report on the programming language Haskell, a non–strict, purely functional language. Version 1.3. Technical report, Yale University, May 1996.

[21] John Peterson, Kevin Hammond, et al. Report on the programming language Haskell, a non–strict, purely functional language. Version 1.4. Technical report, Yale University, April 1997.

[22] Simon L Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *ACM Symposium on the Principles of Programming Languages (PoPL'96)*, St.Petersburg Beach, Florida, January 1996.
URL `ftp://ftp.dcs.glasgow.ac.uk/pub/haskell/glasgow`.

[23] N.J. Radcliffe and P.D. Surry. Formal memetic algorithms. Technical Report EH9 3JZ, Edinburgh Parallel Computing Centre, King's Buildings, University of Edinburgh, Scotland.

[24] M. Vestin. Genetic algorithms in haskell with polytypic programming. Master's thesis, Gteborg university, 1997.
URL `http://www.cs.chalmers.se/~johanj/polytypism/genetic.ps` .

[25] Philip Wadler, editor. *Journal of Functional Programming. Special Issue on State-of-the-art Applications of Pure Functional Programming Languages*, volume 5 (3). Cambridge University Press, July 1995.

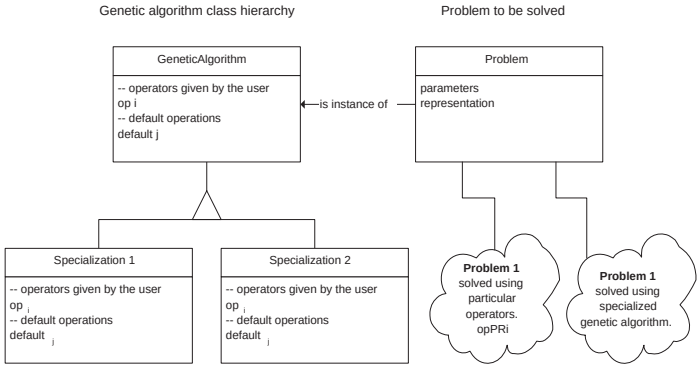


Figure 1: System Architecture

| Population size | Generations | Best found | Reductions | Avg. Fitness |
|---|---|---|---|---|
| 10 | 0 | 6 | 46029 | 18,2 |
| 10 | 10 | 4 | 970766 | 7 |
| 10 | 50 | 2 | 4610193 | 11 |
| 50 | 0 | 0 | 225241 | 2,12 |
| 50 | 10 | 0 | 5030185 | 5,04 |
| 50 | 50 | 0 | 24287463 | 6,12 |
| 100 | 10 | 0 | 10528990 | 8,42 |

Figure 2: MNP - Instance 1

| Population size | Iterations | Best found | Reductions | Average Fitness |
|---|---|---|---|---|
| 10 | 0 | 3 | 83450 | 1,8 |
| 10 | 10 | 4 | 892139 | 4 |
| 50 | 0 | 4 | 413909 | 1,7 |
| 50 | 10 | 5 | 4655925 | 3,82 |
| 100 | 10 | 5 | 9804400 | 3,89 |
| 25 | 50 | 5 | 9921521 | 4,36 |

Figure 3: MB - Instance 1

| Population size | Iterations | Best found | Reductions | Average Fitness |
|---|---|---|---|---|
| 10 | 0 | 4 | 109923 | 3,2 |
| 10 | 10 | 6 | 990434 | 5,6 |
| 10 | 50 | 5 | 4599718 | 4,6 |
| 50 | 0 | 6 | 526777 | 2,66 |
| 50 | 10 | 7 | 5273301 | 4,5 |
| 50 | 50 | 7 | 24074330 | 6,7 |
| 100 | 10 | 6 | 10920553 | 4,92 |

Figure 4: MB - Instance 2

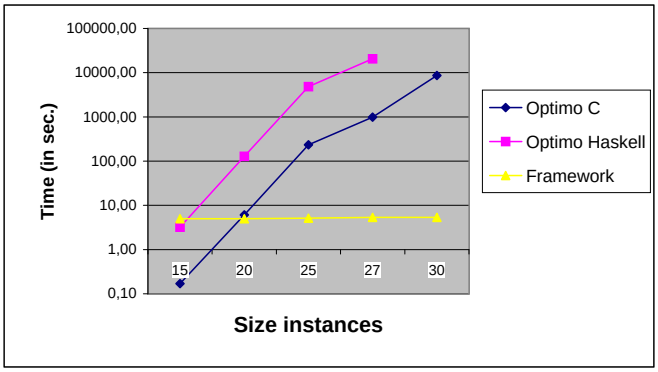


Figure 5: Time comparison